TUTOR

Understanding Data Compression

BY JEFF PROSISE

One of the hottest topics in computer science these days is data compression—the art of shoehorning large amounts of data into small spaces. Storage requirements are growing every day. Ten years ago, a 10MB hard disk could easily accommodate the storage requirements of dozens of programs. Today, if you use Windows and three or four Windows applications, it's hard to get by with less than 80MB. New technologies, such as real-time video playback and digitized sound reproduction, tax the capabilities of today's hardware even more, because a few seconds of uncompressed video or high-resolution sound can tie up several megabytes of storage space. Clearly, the advent of fast, efficient means of compressing data on PCs comes not a moment too soon.

Data compression comes in many forms. Utilities such as PKZIP compress data as a standalone operation so that you can fit more files on a disk or download files from bulletin boards in less time. Disk compression products such as Stacker and DOS 6's new DoubleSpace compress and uncompress data in real time. You may even benefit from data compression without knowing it. Most .PCX and .GIF graphics files, for example, have compression built in. That's why a .PCX or .GIF file is typically much smaller than a TIFF file of the same image. Even Windows .BMP files have an option for built-in compression.

There are two types of data compression: *lossless* and *lossy*. Lossless data compression algorithms ensure that no information is lost in the compression/decompression process; the decompressed version is exactly the same as the original. Lossy compression algorithms give up bit-for-bit accuracy for higher compression ratios. Lossless compression finds application in utilities such as PKZIP and in disk compression software. Lossy compression is typically used to compress video and sound images, where the loss of small amounts of data does not reduce the perceived overall quality of the output. Naturally, lossy compression would be anathema to a disk compression program, because losing a bit here or there could completely alter the meaning of the data.

To the person who's not familiar with common compression algorithms, data compression may seem more like magic than science. To help you understand how this emerging technology works, we'll look at a very simple lossless compression algorithm known as *run-length encoding*. Then we'll briefly discuss the compression algorithms employed by PKZIP, Version 1.1 (ever wonder what PKZIP is really doing when it says "shrinking" or "imploding?") and DOS 6's DoubleSpace.

Find out how PKZIP and DOS 6's DoubleSpace make your files smaller using RLE, Lempel-Ziv-Welch, probabilistic, and Shannon-Fano algorithms.

RUN-LENGTH ENCODING Run-length encoding, or RLE, is one of the simplest and most often used forms of lossless data compression. It works best with files that contain many consecutive occurrences of the same bit pattern, a characteristic that usually applies to bitmapped images.

Here's a simple example illustrating how RLE works. Suppose that you want to compress a file that contains the following 32 bytes of information:

```
00 32 46 00 00 00 00 05
99 02 02 02 01 01 01 01
01 01 01 01 01 01 01 64
00 00 00 00 00 00 00 00
```

If you examine the file closely, you'll notice that it contains several "runs" of bytes with the values 00 and 01. If you listed the file's contents in such a way that runs, or groups of repeating bytes, were listed on the same line, the file would look like this:

```
00
32
46
00 00 00 00
05
99
02 02 02
01 01 01 01 01 01 01 01 01 01 01 01
64
00 00 00 00 00 00 00 00
```

Now imagine what would happen if you replaced each line in this listing with 2 bytes: one identifying the number of bytes in the run, and another identifying the value of the bytes in the run. Written this way, the file would look like this:

```
01 00
01 32
01 46
04 00
01 05
01 99
03 02
11 01
01 64
08 00
```

With this simple form of encoding, you just compressed 32 bytes' worth of data into 20 bytes. Knowing the compression algorithm, a smart program could take the 20 encoded bytes and reproduce the original file by reversing the encoding process. RLE was incorporated into the specification for .PCX and .BMP files because long runs of identical characters are common in bitmapped images. The greater the number and length of repeating runs that a file contains, the higher the compression ratio that RLE can achieve.

There is a downside, however. In the worst case—if a file contained no repeating bytes—this simple form of RLE compression would double the size of the file. More efficient forms of RLE exist (forms that do not expand every nonrepeating byte into 2 bytes), but one must still be careful when choosing a compression algorithm. What's good for one file may not be ideal for another. That's why a well-written compression program must analyze a file before compressing it and must choose the compression method that is best suited to it.

PKZIP PKZIP, probably the world's leading file compression and archiving utility, chooses from three different compression methods—*shrinking, reducing,* and *imploding*—depending on the patterns of data that a file contains. If none of these methods is capable of reducing the file size, PKZIP simply copies the file unchanged to the ZIP file, a process called *storing.*

Shrinking uses a variation of the famous Lempel-Ziv-Welch (LZW) compression algorithm. LZW works by building a dictionary of phrases (groups of one or more characters) from the input stream. When a new phrase is encountered, the compression engine checks to see if that phrase is recorded in the dictionary. If not, the phrase is added to the dictionary and a token that identifies the phrase's position in the dictionary is output. If the phrase was already recorded, then the compressor simply outputs the token for the existing phrase. Whereas RLE lends itself well to sequences of data with repeating single characters, LZW works very well on data with repeating *groups* of characters. The English language uses many repeating character patterns, so LZW is usually a good choice for compressing text files.

One of the factors that affect the efficiency of LZW compression is the length (in bits) of each output token, which in turn determines the maximum size of the dictionary. Tokens are typically from 9 to 15 bits in length. Larger tokens work better for large files, but smaller tokens frequently produce higher compression ratios in small files. PKZIP uses a dynamic sizing method that begins with 9-bit tokens and increases the token size as the size of the dictionary increases, up to a maximum of 15 bits per token. This technique works as well with small files as it does with large files because the final token size is a function of the file size. The chief difference between a classic implementation of LZW and the form that PKZIP uses is that when the dictionary becomes full, PKZIP only partially clears it of phrases. Many LZW compression engines clear the dictionary completely and start building phrases again from scratch.

A well-written compression program must analyze a file before compressing it.

Reducing is actually two algorithms in one. First PKZIP compresses repeating character sequences using an intricate form of RLE. Then it compresses the result using a probabilistic compression method. *Probabilistic compression* is based on the simple premise that you can compress a data stream by using varying numbers of bits to represent different characters (or character patterns) if characters that appear often are assigned the fewest number of bits and characters that appear infrequently are assigned the greatest number of bits. For example, suppose a word processing document contains 1,000 occurrences of the letter E but only 20 occurrences of the letter X. Stored as 8-bit ASCII text, the letters E and X would account for 1,020 bytes. Now suppose that the letter E could be represented in just 4 bits, while the letter X could be represented with 16. Now E and X would account for only 540 bytes—500 bytes for the letter E (1,000 characters at ½ byte per character) and 40 bytes for the letter X (20 characters at 2 bytes per character). Clearly this is a more efficient way to store the data, and would result in a compression ratio of almost 50 percent.

This is a very simplistic illustration of how probabilistic compression works, and it ignores the fact that a probability table must be attached to the compressed data, partially offsetting the gains. PKZIP takes this idea a step further by assigning variable-length bit patterns not to single characters, but to pairs of characters. Also, in order to reduce memory consumption and the size of the probability table, PKZIP limits the table to 32 patterns beginning with a given character. That's fine for text files, because while there may be many occurrences of the characters NE, there will likely be no occurrences of, say, QX. The maximum length of the probability table is 256 (which is the number of characters that can be represented with an 8-bit value) times 32 (the number of combinations supported for each starting character) bytes, or 8K, plus a few bytes extra for overhead. In reality, the table is usually much smaller—especially if the data file itself is not very large.

Imploding also employs two discrete data compression algorithms. First the data is compressed using a form of Lempel-Ziv (LZ) compression called *sliding dictionary.* Sliding dictionary LZ is similar to LZW in that it identifies repeating phrases by outputting tokens that tell the decompression program where in the file the phrase occurred before. But instead of checking the entire file for matching phrases, it uses only part of the file—specifically, a fixed-size block whose address is repeatedly incremented as the file is read. Hence the term "sliding dictionary." PKZIP uses a 4K or 8K block size, which strikes a reasonable balance between compression ratio (the larger the block size, the more the data can be compressed) and compression speed (the larger the block size, the more time it takes to check for repeating patterns). This is a classic form of data compression, and one that, like LZW, lends itself well

to data streams with repeating character sequences.

After it is compressed using the sliding dictionary method, imploded data is then compressed again using *Shannon-Fano trees*. The Shannon-Fano algorithm is a form of minimum redundancy coding that, like the probabilistic compression method described earlier, represents oft-repeated characters with fewer bits than seldom-used characters. Shannon-Fano was one of the very first compression algorithms, developed in the late 1940s by Claude Shannon of Bell Labs and R.M. Fano of M.I.T. The term *tree* comes from the method normally used to implement Shannon-Fano compression, which involves counting the occurrences of each character and constructing a binary tree of 1s and 0s where characters represent "leaves." To determine the numeric code for a given character, you begin at the top of the tree and follow the branches until you reach the specified character. Each branch represents one bit of information. Often-used characters appear closer to the top of the tree, and hence require fewer bits to represent in compressed form.

In December, PKWare introduced PKZIP, Version 2.04g, which replaces the three methods of data compression used in previous versions—shrinking, reducing, and imploding—with a single method called *deflating*. Deflating is similar to imploding except that it uses a larger (32K) dictionary size and a more sophisticated form of Shannon-Fano encoding, resulting in higher compression ratios.

The process of uncompressing a file is usually far easier—and less CPU-intensive—than compressing it. If you'd like to see how PKZIP files are unzipped, you should check out Michael Mefford's PCUNZIP utility, which appeared in *PC Magazine*'s March 31, 1992, Utilities column. You can download it, source code and all, from the Utilities/Tips Forum on PC MagNet, archived as PCUNZP.ZIP. Better yet, you can use it to unzip compressed files downloaded from on-line services.

DOUBLESPACE DoubleSpace—the disk compression driver built into DOS 6—uses a form of sliding-dictionary LZ compression similar to the form PKZIP uses when it "implodes" a file. DoubleSpace uses a 4K sliding dictionary to encode often-repeated phrases as tokens that consist of an offset from the start of the current phrase to an earlier occurrence of the same phrase, and a count byte that identifies the number of characters in the phrase. The greatest difference, of course, between DoubleSpace and compression programs such as PKZIP is that DoubleSpace does all its compression and decompression behind the scenes. DoubleSpace automatically compresses data on its way to a disk and uncompresses it when it is read back. PKZIP, by contrast, must be run manually each time you want a file or group of files compressed.

Borrowing an example from the reviewer's guide that Microsoft distributed to members of the press prior to the release of MS-DOS 6, the phrase

```
therain in Spain falls mainly on
 the plain
```

which requires 43 bytes, would be encoded as

```
the rain <3,3>Sp<9,4>falls
 m<11,3>ly on<34,4>pl<15,3>
```

which requires only 37. Numbers in angle brackets represent tokens; the token <9,4> means "repeat the 4-character sequence that began 9 characters to the left of this token." If you work your way through the encoded text from left to right replacing tokens with literal text, you'll reproduce the original, unencoded message. That's exactly what DoubleSpace does when it expands data read from a compressed volume.

FURTHER READING If you're interested in learning more about data compression methods, there are several books available on the subject. My personal favorite is *The Data Compression Book*, by Mark Nelson (M&T Books, 1992). Unlike many other books on data compression, *The Data Compression Book* is short on equations and long on clear, thoughtfully written explanations. It discusses lossy as well as lossless compression methods, and, for the programmer, it contains plenty of source code examples written in C. □